Data Types, Operators & Bit Manipulations

Topics

1. What is a Data Type?
2. Data Types in C & C++
3. Types of Operators
4. Bit Manipulations
5. Problems & Solutions

What is a Data Type?

Each *variable*[1] in C (or any other language) has an associated data type. Each data type requires different amounts of memory and has some specific operations, which can be performed over it. [[1] Variable is a *name*[2] given to data. [2] Name follows Identifier rules]

Let us briefly describe them one by one:

a. **char**: The most basic data type in C. It stores a single character and requires a single byte of memory in almost all compilers.
b. **int**: As the name suggests, an int variable is used to store an *integer*.
c. **float**: It is used to store decimal numbers (numbers with floating point value) with single precision.
d. **double**: It is used to store decimal numbers (numbers with floating point value) with double precision.
e. **void**: It is a special data type to that does not have a value of any type. One of the uses of void data type is when we have defined functions that just *print* a value, and have no *return* value.

Different data types also have different ranges up to which they can store numbers. These ranges may vary from compiler to compiler. Below is the list of ranges along with the memory requirement and format specifiers on 32-bit gcc compiler

Data Types in C & C++

Data Type / Format Specifier	Memory (Bytes)
short int	2
unsigned short int	2
unsigned int	4
int	4
long int	4
unsigned long long int	8
long long int	8
signed char	1
unsigned char	1
float	4
double	8
long double	12
bool	1

Comparison of a float with a value in C

- `float a = 0.7;`
- uses the double constant `0.7` to create a single precision number (losing some precision), whereas:
- "`if (a == 0.7)`", will compare two double precision numbers (`a` is promoted first).
- The precision that was lost when turning the double 0.7 into the float `a` is not regained when promoting `a` back to a double.
- If you change all those `0.7` values to `0.7f` (to force float conversion, rather than double), or if you just make a double, it will work fine.

You can see this in action with the following code:

```
#include <stdio.h>
int main() {
    float f = 0.7;    // double converted to float
    double d1 = 0.7;  // double kept as double
    double d2 = f;    // float converted back to double

    printf ("double:            %.30f\n", d1);
    printf ("double from float: %.30f\n", d2);
    return 0;
}
```

which will output something like:

```
double:                0.699999999999999955591079014994
double from float:     0.699999988079071044921875000000
```

Important References:

1. **Types of Type Checking (Type System)**
2. **Primitive Data Types in Java**
3. **Internal Data Types in Python**
4. **Representation of a Floating Point Number in Memory**
5. **Data Representation in Memory & Data Conversion**
6. **sizeof(bool) Data Type in C++**

Types of Operators

- **Arithmetic Operators**
  `+, -, *, /, %`
  post-increment (`x++`)
  pre-increment (`++x`)
  post-decrement (`x--`)
  pre-decrement (`--x`)
  These are used to perform arithmetic/mathematical operations on operands.

- **Relational Operators**
  `==, !=, >, <, >=, <=`
  Relational operators are used for comparison of two values. Relational Operator return a *truth value* (true or false) based on comparison of the two operands.

- **Logical Operators** (`&&, ||, !`)

- **Bitwise Operators** (`&, |, ^, ~, >>, <<`)

- **Assignment Operators** (`=, +=, -=, *=,` etc)

- **Conditional (ternary) Operator**:
  Syntax: `(<condition>) ? <true part> : <false part> ;`
  If the given `<condition>` is true, then the `<true part>` is evaluated, else, the `<false part>` is evaluated.

- **Other Operators** (comma[`,`], sizeof, address[`&`], indirecton[`*`], structure-indirection[`->`]) :

Important References

1. **Operator Precedence & Associativity Table in C** [Must Read]
2. **How do Logical Operators work (Short Circuiting)?**
3. **Member Access Operators & Other Operators in C**

Bit Manipulations

1. **& (bitwise AND):** Takes two numbers as operand and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1.
2. **| (bitwise OR):** Takes two numbers as operand and does OR on every bit of two numbers. The result of OR is 1 any of the two bits is 1.
3. **^ (bitwise XOR):** Takes two numbers as operands and does XOR on every bit of the two numbers. The result of XOR is 1 if the two bits are different. XOR is also known as Modulo 2 Sum, i.e., The result of XORing two bits is the same as adding those two bits as a sum, and taking the sum%2.
4. **~ (bitwise NOT):** Takes one number and inverts all bits of it

Truth Table for AND(&), OR(|), XOR(^) and NOT(~)

A	B	&	|	^	~A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

5. **<< (left shift):** Takes two numbers, left shifts the bits of first operand, the second operand decides the number of places to shift.
   - "A << x" implies shifting the bits of 'A' to the left by 'x' positions.
   - The first 'x' bits are generally lost this way. The last 'x' bits have 0. However, in some languages, the first x bits are circulated back to the last x bit positions.
   - Example:
     A = $(29)_{10}$ = $(00011101)_2$ and x = 2, [Considering 8-bit representaion]
     So "A << x" means
     ($(00011101)_2$ << 2) = $(01110100)_2$ = $(116)_{10}$
   - **A << x is equal to multiplication by 2^x**, i.e., "A << x" = $A \times 2^x$.

6. **>> (right shift):** Takes two numbers, right shifts the bits of first operand, the second operand decides the number of places to shift.
   - "A >> x" implies shifting the bits of A to the right by x positions.
   - The last x bits are lost this way.
   - Example :
     A = $(29)_{10}$ = $(00011101)_2$ and x = 2,
     So "A >> x" means
     ($(00011101)_2$ >> 2) = $(00000111)_2$ = $(7)_{10}$
   - **A >> x is equal to division by 2^x**, i.e., "A >> x" = $A/2^x$.

Bit Tricks

- `x & (x-1)` : will clear the least significant set bit of **x**.
- `x & ~(x-1)` : extracts the least significant set bit of **x** (all others are clear). Pretty patterns when applied to a linear sequence.
- `x | (x+1)` : sets the least significant unset bit of **x**
- `x | ~(x+1)` : extracts the least significant unset bit of **x** (all others are set).

Important References

1. **Topcoder: Fun with Bits**
2. **Important Notes on Bit Manipulation**
3. **Must Know Low Level Bit Hacks**
4. **Bit Magic (GeeksforGeeks)**
5. **XOR and Modulo 2 Sum**

Problems & Solutions

1. Check if i^{th} bit is set in a number **N**. [Note: LSB is considered as 0^{th} bit]
   Solution: `(N & (1 << i)) != 0` [or] `((N >> i) & 1) != 0`

2. Check if a given number **N** is power of 2 or not, i.e., **N** = $2^{something}$ or not.
   Solution: `N & (N - 1) == 0`

3. Check if two given signed integers **a** and **b** are of the same sign.
   Solution: `(a ^ b) >= 0`

4. Write a program to print binary representation of a given number.
   Solution: Extract bit by bit - loop or recursion

5. You are given an array with size *n*, which contains elements from range 1 to *n*. One element is repeated and one element is missing. Find these two elements.
   Solution:
   a. Mathematical: Compare sum and product of array elements with sum and product of first n natural numbers. 2 unknowns and 2 equations - Solve
   b. Using XOR
      i. Let **a** and **b** be the two numbers to be found.
      ii. XOR the given array with the numbers 1...*n*, call it **X**.
      iii. You have: `X = a ^ b`.
      iv. Extract any set bit from **X**. The bit position indicates that this bit is different in **a** and **b**.
      v. Based on this bit position, separate the array into two parts - One having numbers with this bit set and the other having numbers with this bit unset. Call this `SetA-0` and `SetA-1`.
      vi. Do step(v) for the numbers in the range 1...*n* and call them `SetB-0` and `SetB-1`.
      vii. `a = SetA-0 ^ SetB-0; b = SetA-1 ^ SetB-1;`

viii. Check for which is missing and which is repeated, using the array.

6. Find the number of set bits in a given integer.
   Solution: `(N & (N-1))`
   `//Unsets the Least Significant Set Bit -- repeat until N becomes 0.`

7. Compute **`pow(int a, int N)`** *i.e.*, compute a^N using fast exponentiation.
   Solution:

```
for(int i = 0; i <= log2(N); i++) {
      if(N & (1 << i)) {
            ans = ans * a;
      }
      a = a * a;
}
//Hint: log2(N) = Position of Most Significant Set Bit in N.
```

8. Given two bit positions: **x** & **y**, return a number with $\mathbf{x}^{th}$ and $\mathbf{y}^{th}$ bits being set.
   Solution: `(1 << x) | (1 << y)`

9. Construct a number with x set bits, followed by y unset bits, for `1 <= x,y <= 10`.
   Example: `x = 3, y = 5` => $(11100000)_2$
   `x = 2, y = 1` => $(110)_2$
   Solution: `((1 << x) - 1) << y` [or] `(1 << (x+y)) - (1 << y)`
   `// (2^x-1)*2^y`